IUT Orsay

DUT Informatique
Module Système S4-L
2010 / 2011

Travaux Pratiques nº 5 : Threads

Nom(s) :
Groupe :
Date :

Objectifs : apprendre à créer, travailler avec et arrêter des threads (ou processus légers). Savoir reconnaître les données partagées entre différents threads. Être capable d'orchestrer la synchronisation de threads au moyen des primitives de terminaison ou de sémaphores d'exclusion mutuelle.

1 Avant propos : parallélisation d'applications

Jusque très recemment, l'amélioration des performances des architectures était principalement liée à l'augmentation de la fréquence des processeurs. L'optimisation des programmes pour les ordinateurs grand public (en simplifiant) était majoritairement une affaire de choix d'algorithmes, d'optimisation des accès à la mémoire et d'utilisation d'opérations moins coûteuses ou mieux ordonnées pour les processeurs. Depuis 2006 et l'apparition des premiers microprocesseurs multi-cœurs pour les ordinateurs personnels, le gain de performance passe par l'utilisation simultanée et efficace des différents cœurs (comme depuis toujours sur les architectures haute performance). La programmation par threads est depuis longtemps et toujours aujourd'hui un des meilleurs moyens de tirer parti de ces architectures.

Le but de ce TP est de paralléliser une application existante afin d'en améliorer les performances. Malheureusement les salles machines de l'IUT ne sont pas encore nombreuses à être équipées d'ordinateurs multi-cœurs. Il est donc probable que le TP nécessite un certain effort d'imagination pour observer un quelconque gain de performances ! Cependant le travail réalisé sera effectivement celui d'une parallélisation de programme authentique.

2 Parallélisation d'un moteur 3D

La parallélisation d'une application avec des threads n'a d'intérêt que pour une application demandant de lourds calculs sur une architecture capable de tirer profit du parallélisme. Nous prendrons l'exemple d'un moteur 3D s'exécutant sur un processeur disposant de six cœurs (c'est par exemple le cas du processeur IBM Cell équipant la console de jeux PlayStation 3 qui dispose de huit *synergistic processing elements* mais dont seulement six sont disponibles pour les programmes de jeux).

Récupérez le code de l'application de moteur 3D sur la page du Système S4 :

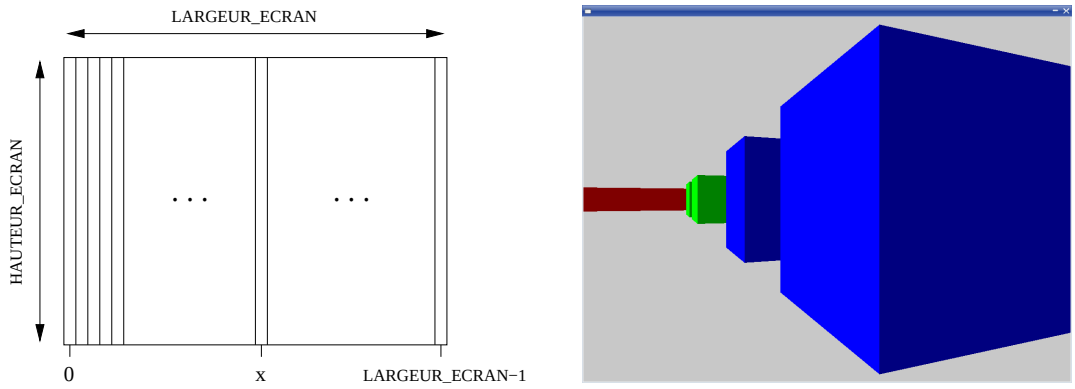
`http://www.lri.fr/~bastoul/teaching/systeme`

Compilez l'application à l'aide de la commande `make` puis lancez l'application par `./moteur3D`, la fenêtre d'affichage du moteur devrait alors afficher (voir Figure 1(b)). Vous pouvez vous déplacer dans la scène en utilisant les touches directionnelles et quitter l'application en appuyant sur *Esc* ou *Q*. Le nombre de frames par secondes moyen est affiché périodiquement sur la sortie standard.

Le cœur du programme est une simple boucle qu'on trouvera dans la fonction `main()` dans le fichier *main.c* (c'est le seul fichier qu'on aura à modifier durant ce TP). Son code est le suivant :

```
/* Boucle de calcul et d'affichage des colonnes de pixels */
for (x = 0; x < LARGEUR_ECRAN; x++)
  moteur_dessiner_colonne(x,LARGEUR_ECRAN,HAUTEUR_ECRAN);
```

Cette boucle réalise le calcul d'une nouvelle image, colonne de pixels par colonne de pixels. L'appel à la fonction `moteur_dessiner_colonne()` réalise le calcul et le dessin de la colonne de pixels d'abscisse `x`. Le découpage de l'écran en colonnes de pixels est illustré en Figure 1(a). Durant l'exécution de cette boucle, chaque itération (c'est à dire le calcul et le dessin de chaque colonne de pixels) peut se faire indépendamment des autres itérations. En effet, chaque itération a besoin d'informations telles que le tableau `carte` (et d'autres variables qui ne sont pas déclarées dans le fichier *main.c*), mais aucune de ces variables n'est modifiée lors du calcul et de l'affichage d'une colonne de pixels. On appelle ce type de boucle des *boucles parallèles*. On va chercher à répartir dans différents threads les calculs et affichages de colonnes de pixels.



(a) Découpage de l'écran en colonnes (b) Image composée de 1024 colonnes

FIGURE 1 – Dessin de l'image par colonnes de pixels

Pourquoi les threads sont bien plus adaptés que les processus pour exploiter le parallélisme de cette application ?

Combien de threads au maximum pourrait-on créer pour cette application ? Pourquoi ne serait-ce pas une bonne idée d'en créer autant ?

Réalisez une première version parallèle du moteur 3D qui répartira les calculs dans **deux** threads. Le code du fichier *main.c* subira encore des modifications avant d'être rendu.

Les calculs de ce moteur 3D, même s'il n'est pas du tout optimisé, sont trop simples pour poser problème aux machines actuelles. Vous pouvez utiliser la variable globale `ralentissement` pour ralentir les calculs. Cherchez la bonne valeur de `ralentissement` pour diviser par cinq la performance du moteur 3D original (sans threads) en frames par secondes.

Comparez la version originale du moteur 3D et la version multi-thread avec la même valeur de variable `ralentissement` que vous avez trouvé. Donnez les valeurs en frames par seconde (ne faire aucun déplacement dans la scène car la performance est aussi fonction de l'image à afficher) des deux moteurs. Qu'en concluez-vous sur les capacités multi-threads de votre système ou/et ordinateur ?

Réalisez une seconde version parallèle du moteur 3D qui répartira les calculs dans **six** threads. Comparez une nouvelle fois les performances. Qu'en concluez-vous ?

Vous pouvez visualiser le besoin en ressources de votre application et des threads de votre application à l'aide de la commande `top`.

Avec quelles options appelez-vous la commande `top` pour ne visualiser que votre application décomposée en threads (pour vous aider, voir la page de man de cette commande) ?

3 Synchronisation de threads

On souhaite que les couleurs des murs de la scène 3D changent périodiquement toutes les cinq secondes. Pour cela, il suffit simplement de modifier la valeur des entrées non nulles dans le tableau `carte`. Pour réaliser une opération périodique on déroutera le signal `SIGALRM`. La difficulté est que l'utilisation de la primitive `signal()` n'est pas possible conjointement à l'utilisation de threads. Il faut à la place utiliser la primitive `sigaction()`, qui est plus puissante mais de manipulation plus complexe. En résumé (car l'utilisation des *signaux Posix* n'est pas le but de ce TP), il faut déclarer et remplir une structure `struct sigaction` qui indiquera ce qu'il faut faire en cas de réception d'un signal, puis utiliser la primitive `sigaction()` pour faire le lien entre un signal et les informations contenues dans la structure. Voici un squelette pour dérouter le signal `SIGALRM` vers la fonction `traiter_sigalrm()` :

```
#include <signal.h>
...
void traiter_sigalrm(int signo) {
...
}
...
  /* Deroutement du signal SIGALRM avec sigaction() */
  struct sigaction s;
  s.sa_handler = traiter_sigalrm;
  sigemptyset(&s.sa_mask);
  s.sa_flags = 0;
  sigaction(SIGALRM,&s,NULL);
  /* Ensuite on peut utiliser alarm() classiquement */
...
```

Un thread sera dédié au travail de modification périodique des couleurs.

Étudiez le besoin en synchronisation engendré par la modification du tableau `carte`. Que peut-il arriver sans synchronisation ? Si on souhaite éviter cette situation, comment s'y prendre (attention à préserver le parallélisme) ?

Réalisez une dernière version parallèle du moteur 3D en reprenant la précédente et en y ajoutant le nouveau thread dédié à la modification des couleurs. **Vous joindrez le code du fichier *main.c* à votre compte-rendu de TP.**

Commentaires personnels sur le TP (résultats attendus, difficultés, critiques etc.).